Starting today: Algorithms

- previously: TMs capture “universal computation”
- next few weeks: do things on a universal computer
  - but which ones? and what are we doing on them?

- Formally, an algorithmic problem is the task of computing some function \( f : \Sigma^* \rightarrow \Sigma^* \) (restricted output to \( \{0,1\} \))

- input \( w \in \Sigma^* \) is an encoding of a “valid input”
- output \( x \in \Sigma^* \) is also an encoding

on algorithm is... some kind of “program” \( A \) s.t.
\[ A(w) = f(w) \quad \forall w \in \Sigma^* \]

What (Turing-complete) model will we assume?
- Unit-cost RAM model
  - basic data type is an integer
  - numbers fit into “words”
  - arithmetic/comparison on words take constant time
    - bitwise ops/flips, ceilings require some care.
  - arrays allow random access
  - pointers store addresses in words

Caution:
- sometimes we have situations where unit-cost makes sense
  e.g., analyze arithmetic in terms of all \#bits
- assumptions only valid if algo’s do not produce overly large intermediary values.
  - large enough numbers need to broken up into multiple words
When all else fails, fall back to TMs.

Reductions \( A \leq B \)

Informally, given an instance of problem \( A \)
convert it into an instance of problem \( B \)
apply known algorithm to problem \( B \)
convert output into correct solution for problem \( A \)

This is a very powerful algorithms design technique.

Instead of reinventing a tool, use someone else's work.

Quite often in this class the easiest way to come up
with an algorithm is to reduce your problem to another
problem with existing algorithm.

Example: given an array of integers, are there any duplicates?

Naive algorithm: double for loop:

\[
\begin{align*}
&\text{for } i \text{ from 1 to } n \\
&\quad \text{for } j \text{ from 1 to } n:\ \\
&\quad \text{if } A[i] = A[j] \\
&\quad \quad \text{return true}
\end{align*}
\]

Better idea: reduce to sorting.

\[
\begin{align*}
&\text{sort } A \\
&\text{for } i \text{ from 1 to } n-1 \\
&\quad \text{if } A[i] = A[i+1] \\
&\quad \quad \text{return true}
\end{align*}
\]

Next 2.5 weeks: special kind of reduction

called Recursion
Next 2 1/2 weeks: special kind of reduction called **recursion**

we did a lot of recursion during automata, hopefully you know by now:

**Recursion** = induction

Recursion as an algorithmic technique:
reduce the problem of solving some instance of this problem
to the problem of solving a smaller instance of
the same problem

---

**Towers of Hanoi**

---

allowed to:
- move one disk at a time
  - (only the top disk)
- put smaller disks on top of larger ones

**goal:** start w/ a stack of disks on peg 
& move it to another peg

if \( n > 0 \):

\[
\text{Hanoi}(n-1, \text{src}, \text{tmp}, \text{dst})
\]

move disk \( n \) from \( \text{src} \) to \( \text{dst} \)

\[
\text{Hanoi}(n-1, \text{tmp}, \text{dst}, \text{src})
\]

---

Running time of this algo? (Can't just move)

\[
T(n) = \# \text{ of moves to get } n \text{ disks from } \text{src} \text{ to } \text{dst}.
\]
\[ T(n) = \begin{cases} 
0 & \text{if } n = 0 \\
T(n-1) + 1 + T(n-1) & \text{if } n > 0.
\end{cases} \]

Guess \( T(n) = 2^n - 1 \).

<table>
<thead>
<tr>
<th>Base case: ( n = 0 ), ( 2^0 - 1 = 0 ).</th>
<th>Assume for ( k &lt; n ) that ( T(k) = 2^k - 1 ).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assume for ( k &lt; n ) that ( T(k) = 2^k - 1 ).</td>
<td>Assume for ( k \leq n ) that ( T(k) = 2^k - 1 ).</td>
</tr>
<tr>
<td>( T(n) = 2T(n-1) + 1 ) def. ( = 2(2^{n-1}) + 1 )</td>
<td>Two cases:</td>
</tr>
<tr>
<td>( = 2^n - 1 ) math. \</td>
<td>( n = 0 ): ( T(0) = 0 = 2^0 - 1 )</td>
</tr>
<tr>
<td>\</td>
<td>( n &gt; 0 ): (same as left side)</td>
</tr>
</tbody>
</table>

Conclusion: \( T(n) = O(2^n) \)

---

**Merge Sort**

Input: \text{SORTINGEXAMPLE}\n
Divide: \text{SORTINGEXAMPLE}\n
Recurse Left: \text{INORST}\n
Recurse Right: \text{AGECLMPX}\n
Merge: \text{AEGLMNOPRSTX}\n
\[
\text{MergeSort}(A[1..n]) \]

\[
\begin{cases} 
\text{if } n > 1 \\
\text{MergeSort}(A[1..n]) \\
\text{MergeSort}(A[m+1..n]) \end{cases}
\]

\[
\text{Merge}(A[1..n], m) \]

\[
\text{recursion funny!} \text{ recursion funny!} \]

\[
\text{reduced to the problem of merging two sorted arrays.}
\]
Merge:  

if \( X \) empty: 
  set \( Z \) to be \( Y \).

routine of merge (exercise).

\[
T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + O(n) \\
\text{(justification for ignoring } \frac{n}{L} \text{ for Big O)}
\]

- see Jeff's book

Recursion Tree:

"extrawork": 
- (non-recursive part)
- of \( T(n) \)
- (recursive calls in \( T(n) \))

\[
\sum_{l=0}^{H} \left( \text{total work at level } l \right)
\]

+ base case work

level \( l \) corresponds to \( T\left(\frac{n}{2^l}\right) \)

at level \( H, T\left(\frac{n}{2^H}\right) \) is a base case

\( H = \text{const.} \) solve for \( H \rightarrow H = O(\log n) \)
Worst case analysis of Quicksort

\[ T(n) = O(n) + \max_r \left( T(r-1) + T(n-r) \right) \]
\[ \leq O(n) + T(0) + T(n-1) \]

- At level \( l \): \( T(n-l) \).
- At level \( H \): \( n-H = \text{const} \)
  So \( H = O(n) \).

\[ T(n) = O(n^2) \]

\[ T(n) = T(\lceil n/2 \rceil) + 1 \]

- At level \( l \): \( T(n^{1/2^l}) \)
  \( H: n^{1/2^H} = \text{const} \)
  \( \log(n^{1/2^H}) = \log(\text{const}) \)
  \( \Rightarrow 2^{-H} \log n = \text{konst} \)